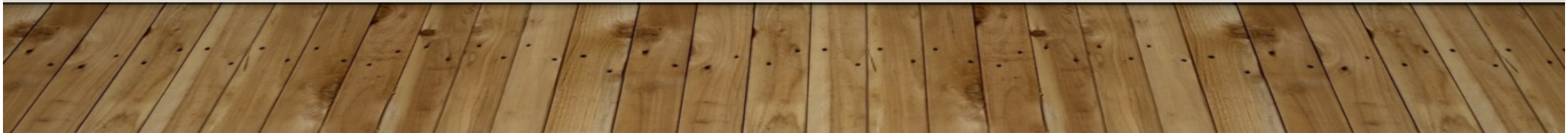
OPERATING SYSTEMS
CS3500

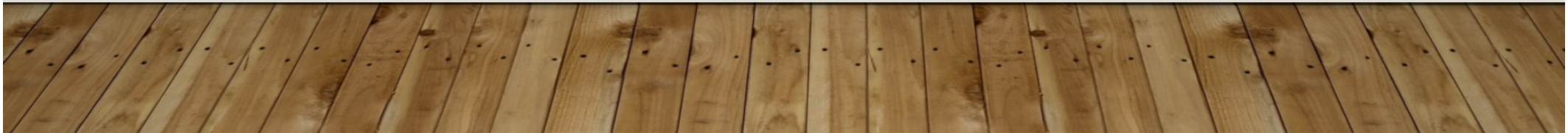
PROF. SUKHENDU DAS DEPTT. OF COMPUTER SCIENCE AND ENGG., IIT MADRAS, CHENNAI – 600036.

Email: sdas@cse.iitm.ac.in
URL: http://www.cse.iitm.ac.in/~vplab/os.html

OCT. – 2022.

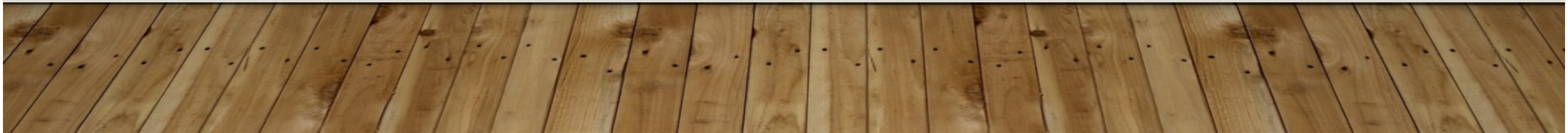


VIRTUAL MEMORY - I



OUTLINE

- Background
- Demand Paging
- Copy-on-Write



BACKGROUND

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

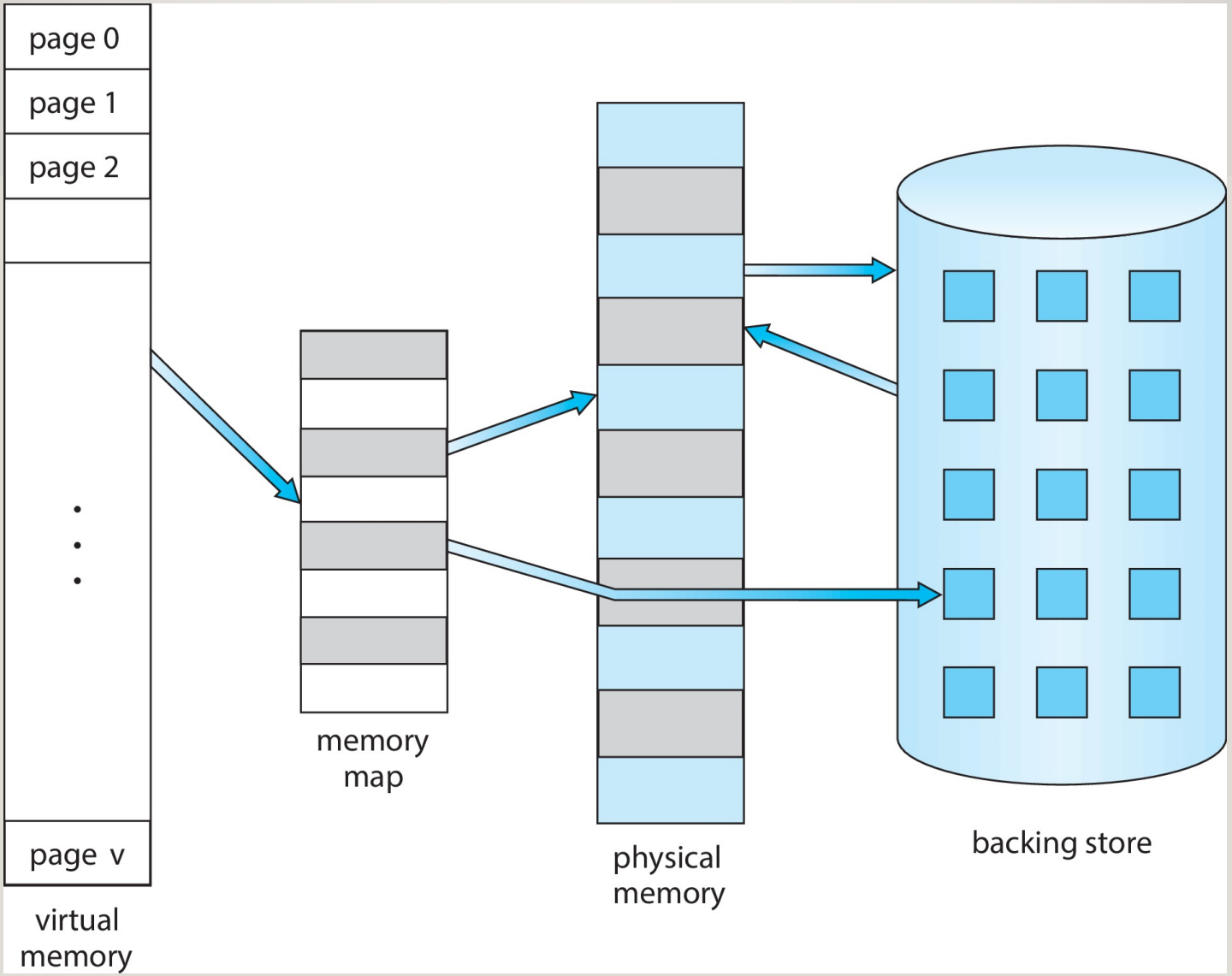
VIRTUAL MEMORY

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

VIRTUAL MEMORY (CONT.)

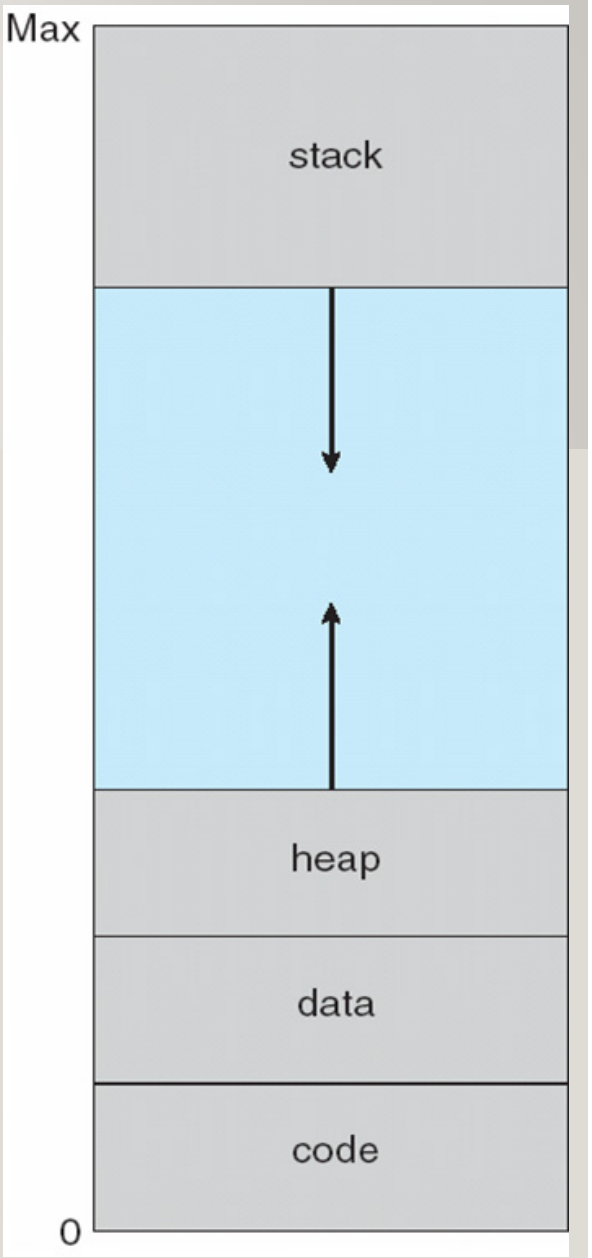
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

VIRTUAL MEMORY THAT IS LARGER THAN PHYSICAL MEMORY

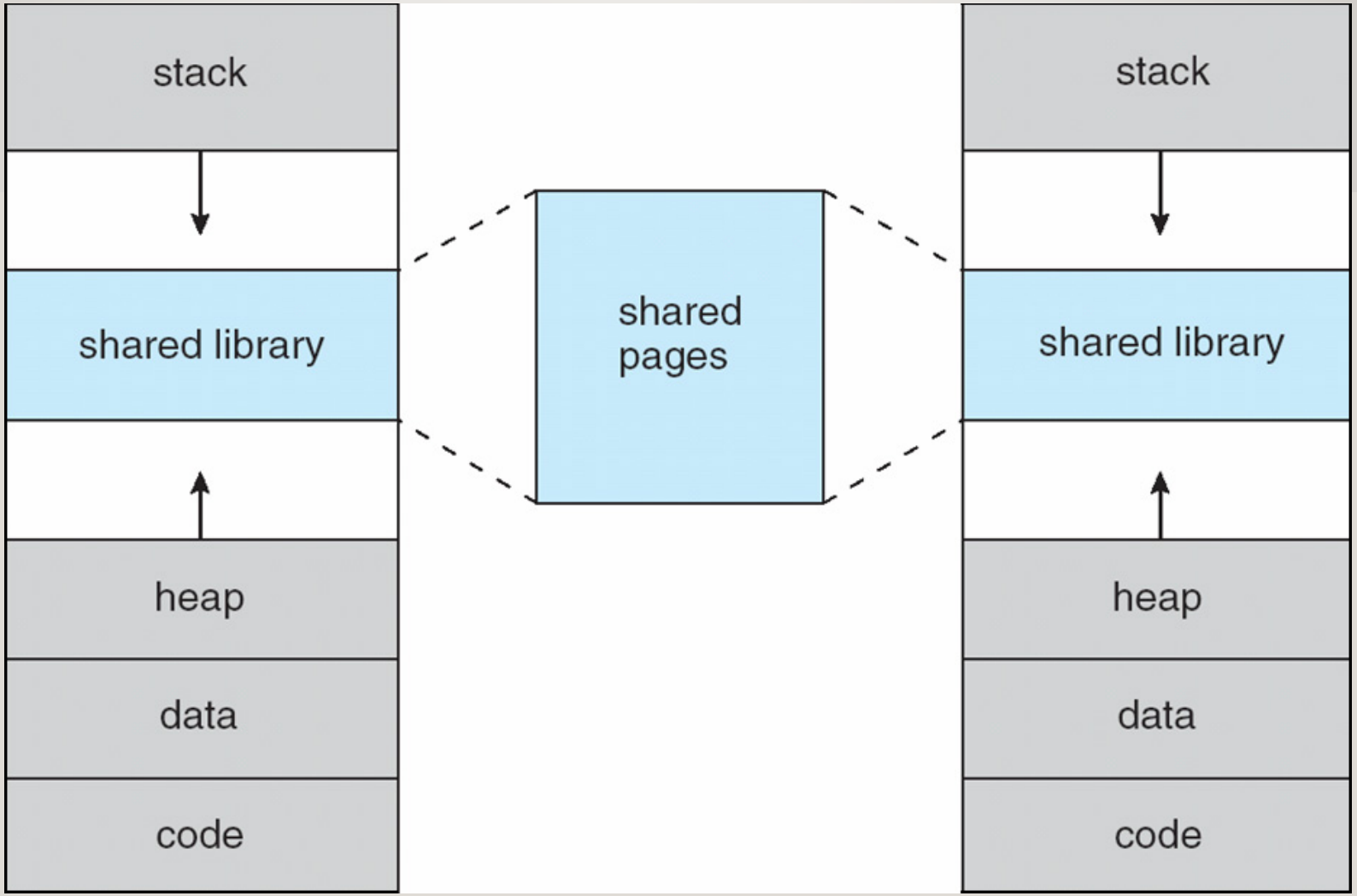


VIRTUAL-ADDRESS SPACE

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



SHARED LIBRARY USING VIRTUAL MEMORY

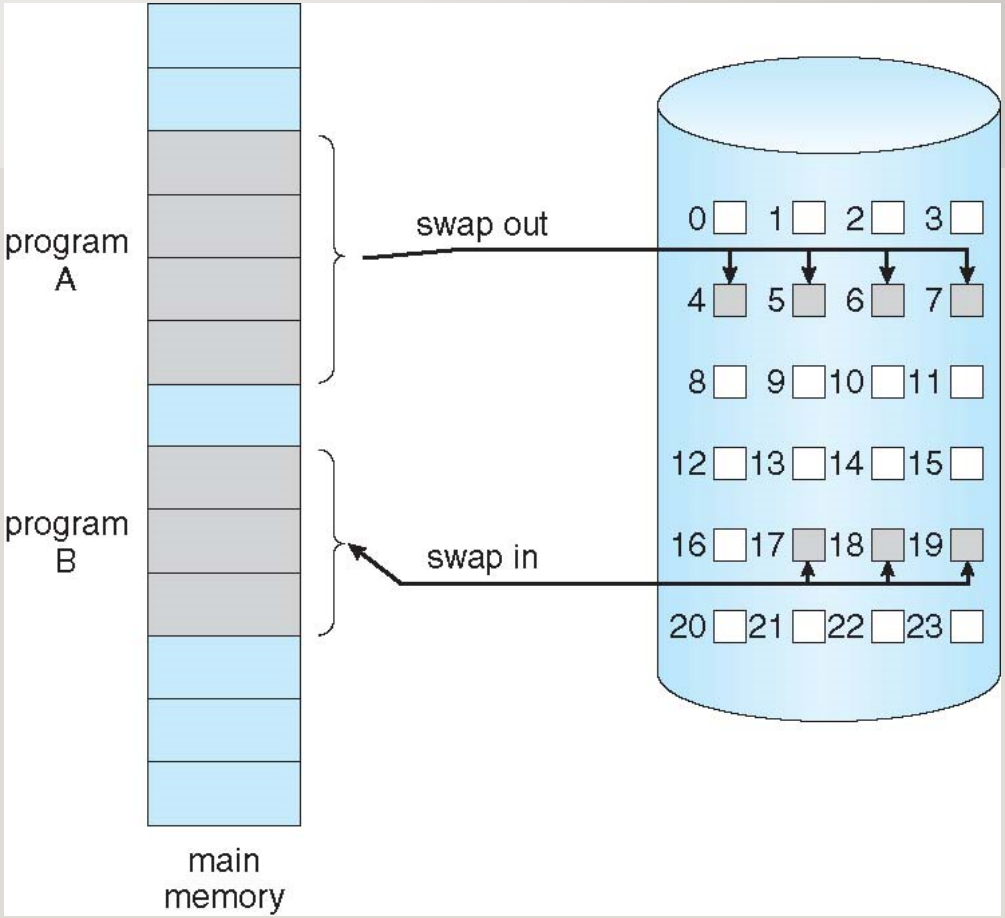


DEMAND PAGING

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on next slide)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

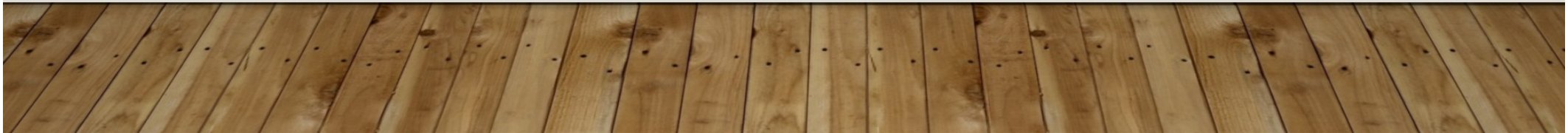
DEMAND PAGING (CONT)

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)



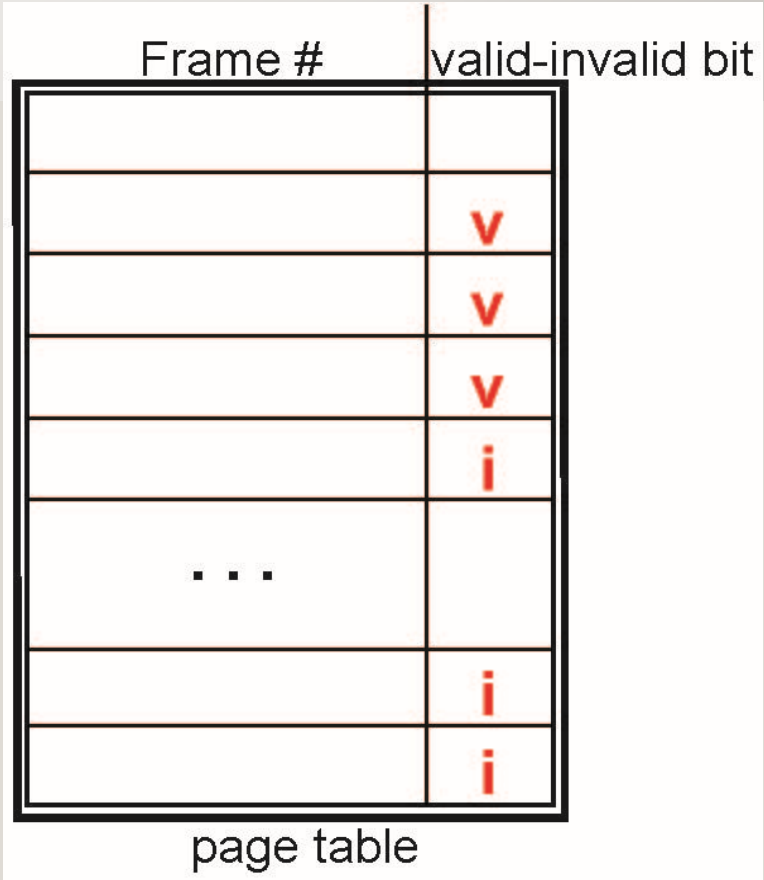
BASIC CONCEPTS

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code



VALID-INVALID BIT

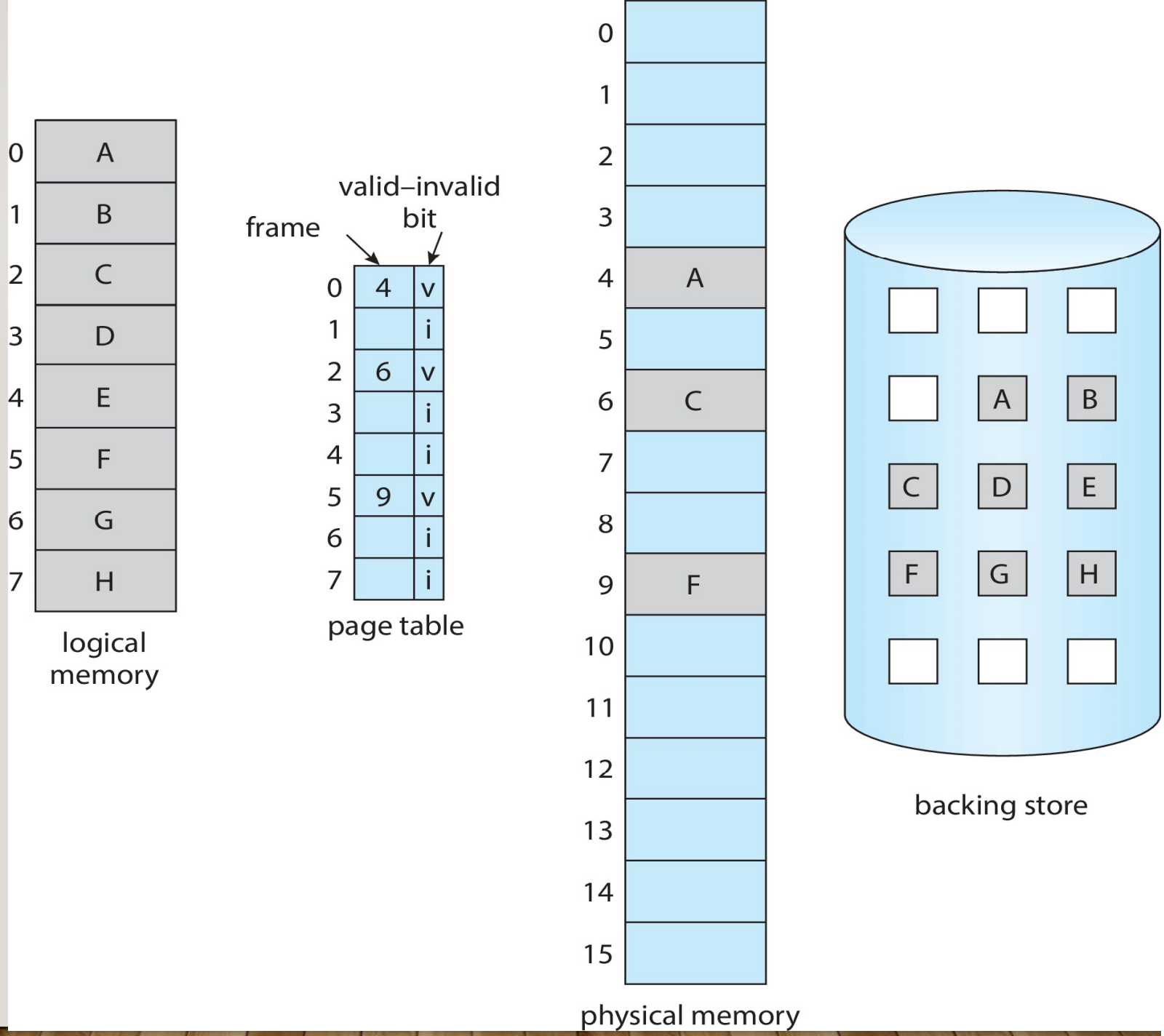
- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory – **memory resident;** **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is
  i ⇒ page fault

PAGE TABLE

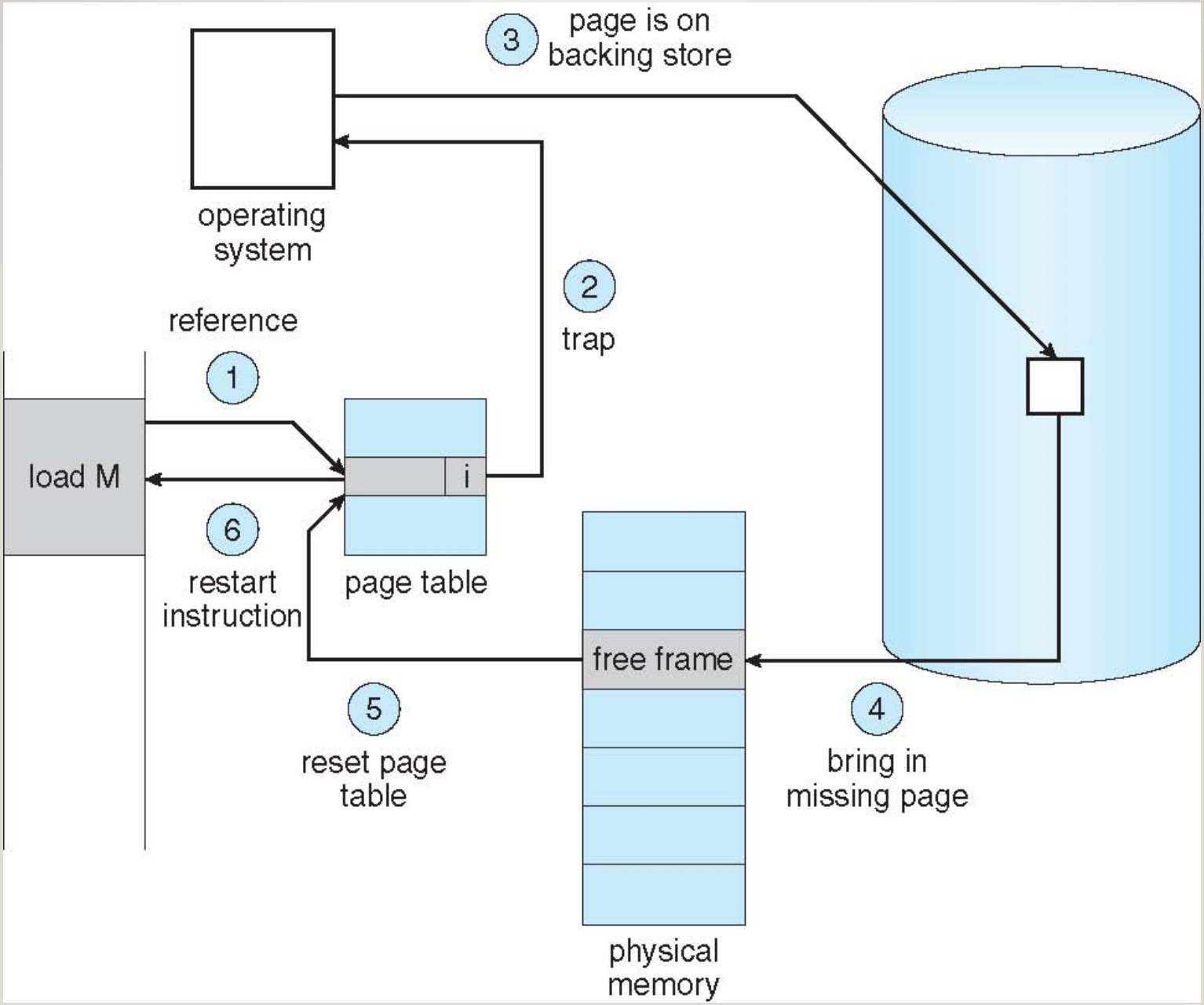
WHEN SOME PAGES ARE NOT IN MAIN MEMORY



STEPS IN HANDLING PAGE FAULT

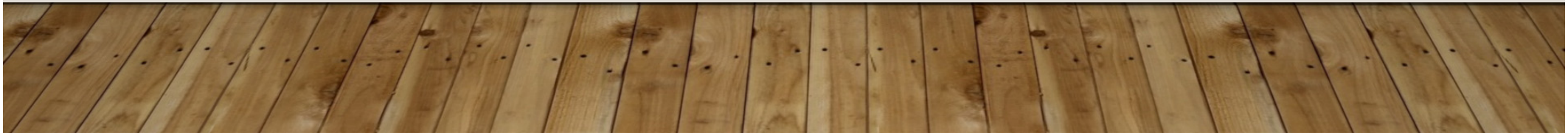
1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

STEPS IN HANDLING A PAGE FAULT (CONT.)



ASPECTS OF DEMAND PAGING

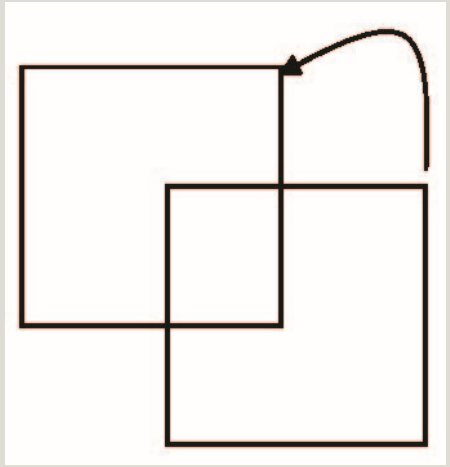
- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And so on, for every other pages of the process on first access
  - **Pure demand paging -** never bring a page into memory until it is required
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart



INSTRUCTION RESTART

MVC TARGET+10(120), SOURCE+3 //MOVE 120 BYTES STARTING AT SOURCE+3 TO TARGET+10

- Consider an instruction that could access several different locations
  - Block move (either block straddles the page boundary) – say, page fault after partially written



- Auto increment/decrement location
- Restart the whole operation?
  - What if source and destination overlap?

1. ***Fetch and decode the instruction (ADD).***
2. ***Fetch A.***
3. ***Fetch B.***
4. ***Add A and B.***
5. ***Store the sum in C.***

(if accessing "C", generates *page fault (PF)* - The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again.

What is PF occurs at step 3 ?

Instruction Restart

- **Auto increment/decrement location**

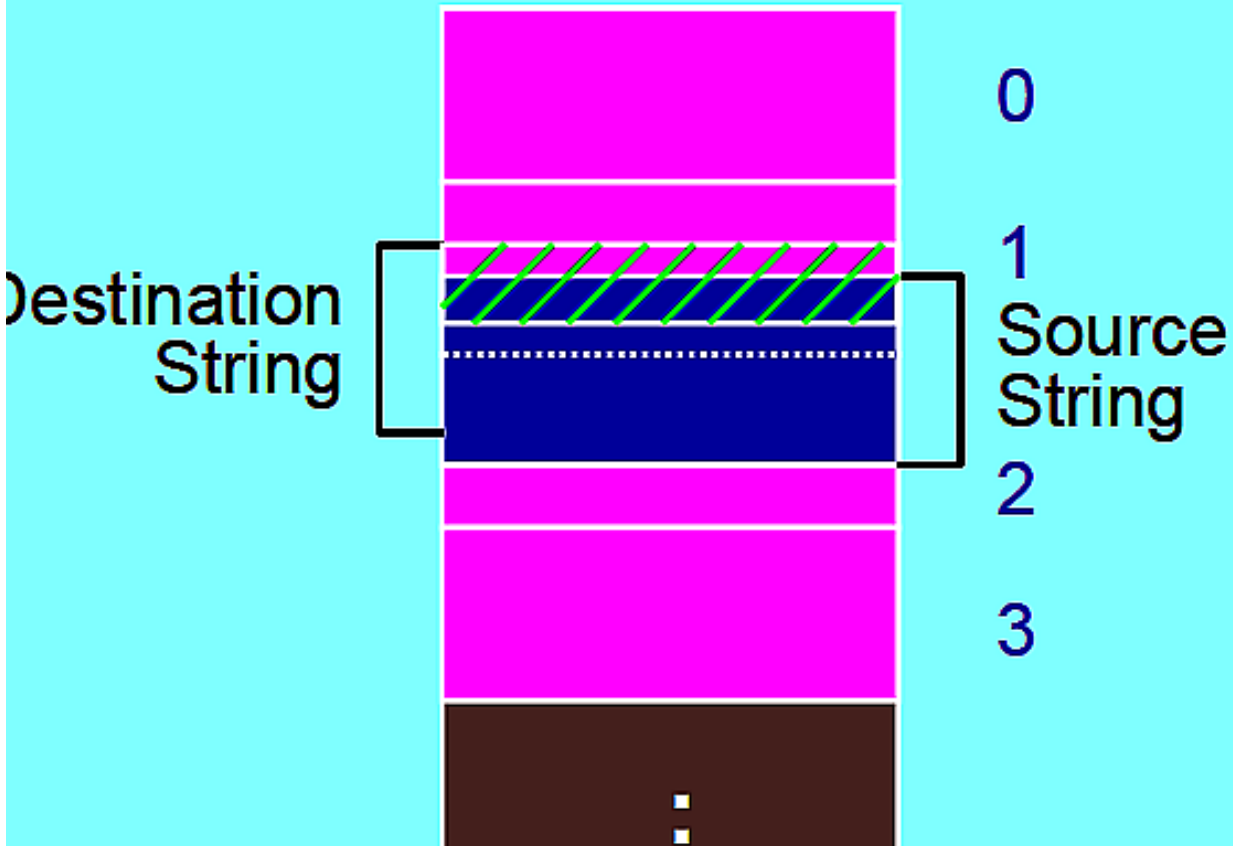
→Special addressing mode (special case of register indirect addressing)
→Example : PDP-11
→Use a register as a pointer and automatically decrement or increment the register as indicated.
→ Instruction : ***MOV (R2)+, -(R3)*** /*copies the contents of the location pointed to by register 2 into the location pointed to by register 3. */

→Register 2 is incremented after it is used as a pointer; register 3 is decremented (by 2) ***before*** it is used as a pointer.

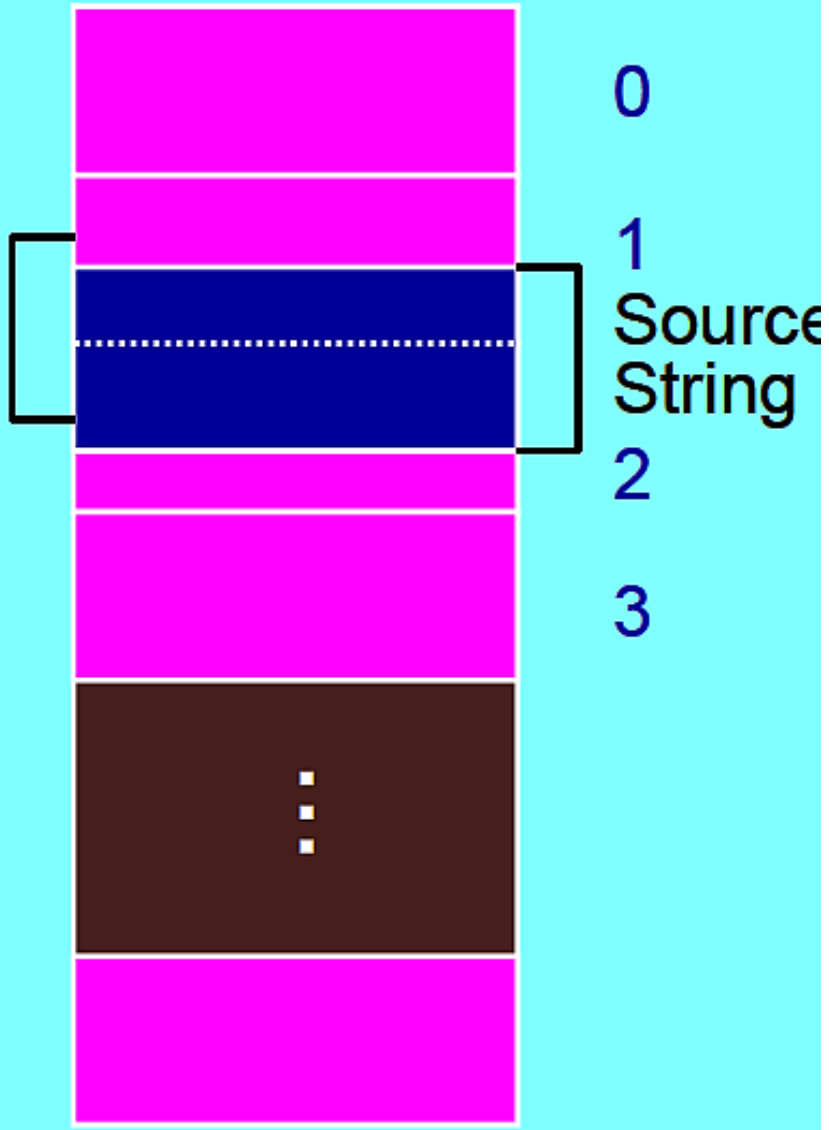
→A fault occurs when trying to store into the location pointed to by register 3
→To restart the instruction, we must reset the two registers to the values they had ***before*** we started the execution of the instruction.

One solution is to create a new special status register to record the register number and amount modified for any register that is changed during the execution of an instruction. This status register allows the operating system to "undo" the effects of a partially executed instruction that causes a page fault

Block Move



Block Move



FREE-FRAME LIST

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.

- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

STAGES IN DEMAND PAGING – WORSE CASE

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    a) Wait in a queue for this device until the read request is serviced
    b) Wait for the device seek and/or latency time
    c) Begin the transfer of the page to a free frame

STAGES IN DEMAND PAGING (CONT.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

PERFORMANCE OF DEMAND PAGING

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$\text{EAT} = (1 - p) \times \text{memory access} + p\ (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$

DEMAND PAGING EXAMPLE

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

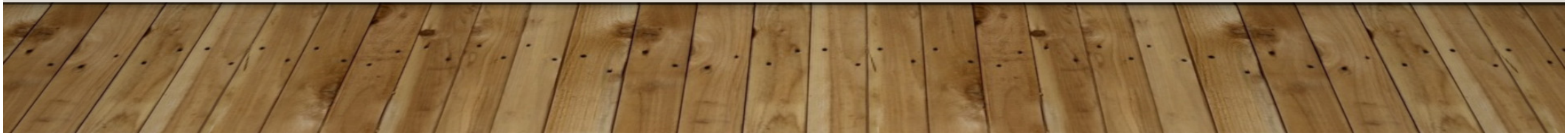
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

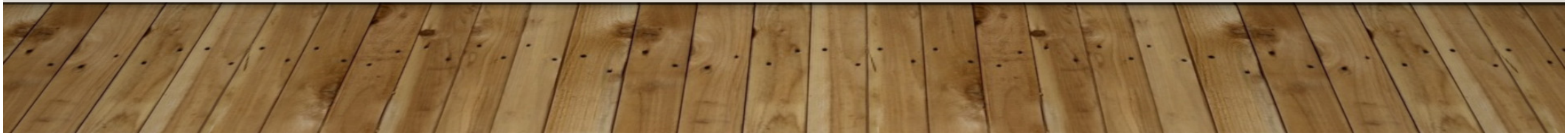
DEMAND PAGING OPTIMIZATIONS

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

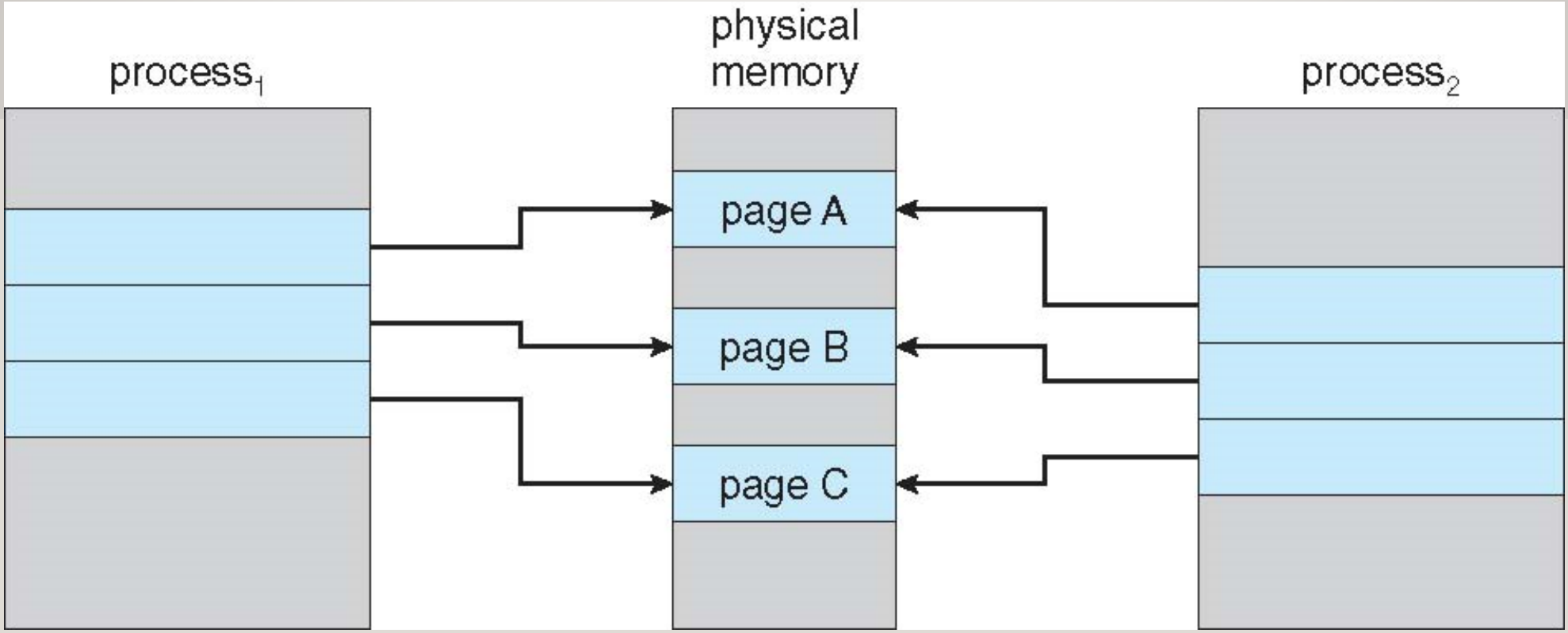


COPY-ON-WRITE

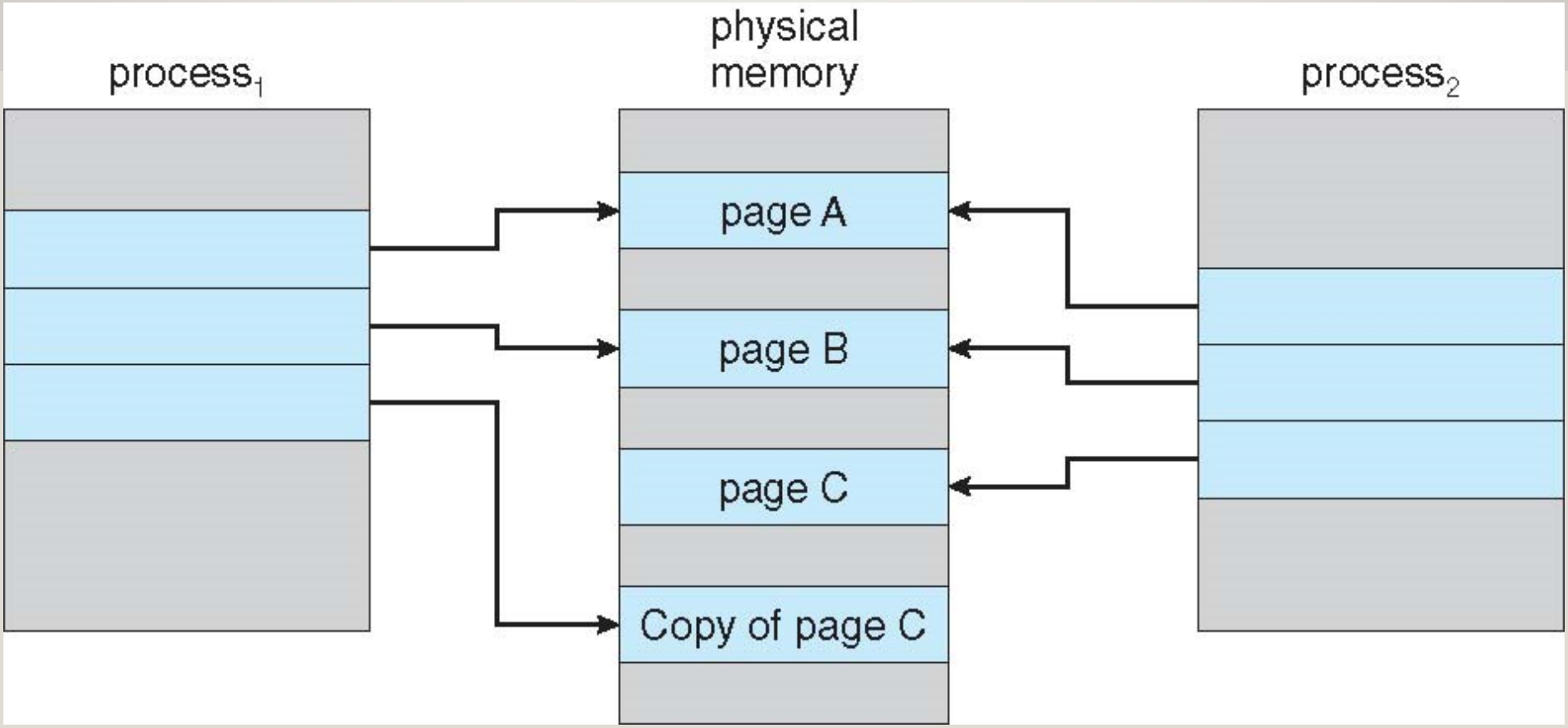
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory; no duplicate pages in P/M
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it? – security, reading garbage.
- `vfork()` variation on `fork() - not (fork + COW);` system call suspends the parent and child uses address space of parent (indirect COW)
  - Designed to have child call `exec(); use with caution`
  - Very efficient (- used in cmd line shell interface)



BEFORE PROCESS *1* MODIFIES PAGE C



AFTER PROCESS 1 MODIFIES PAGE C



Do you set dirty bit after zeroing in ?